Rogue Decision Procedures

Aaron Stump, Arumugam Deivanayagam, Spencer Kathol, Dylan Lingelbach, and Daniel Schobel

Dept. of Computer Science and Engineering
Washington University in St. Louis
One Brookings Drive, Campus Box 1045
St. Louis, MO 63130-4899

Corresponding email: stump@cs.wustl.edu

Abstract. Efficient decision procedures require a substantial engineering effort to implement in mainstream languages. This paper proposes a new programming language called Rogue for implementing decision procedures. Rogue is a rewriting language with backtrackable mutable expression attributes and an interface to a fast SAT solver. Work in progress on implementing a Nelson-Oppen style cooperating validity checker in Rogue is also briefly described.

1 Introduction

High-performance decision procedures (DPs) for decidable logical theories have enjoyed significant recent research and development activity, largely due to their usefulness for system verification. But developing high-performance DPs, particularly the combinations of cooperating DPs most useful for verification, is a significant engineering challenge. This is largely because of the following common characteristics of most DPs:

symbolic computation Most DPs involve some form of manipulation of expressions. This often takes the form of rewriting. For example, all DPs which follow the Shostak approach [13] of canonizing terms and solving equations are based on rewriting expressions.

imperative data structures DPs usually employ special-purpose data structures for efficiency. Implementing such data structures in a programming language is most natural if the language has mutable references or some other imperative construct.

interface to SAT solver For many DPs, it is important to be able to take advantage of the great advances that have been made in the last decade in the performance of propositional SAT solvers. Early empirical results suggest that for highest performance, a tight coupling of the SAT solver and the rest of the DP may be necessary [3, 9]. Hence, some kind of interface to a fast SAT solver is needed.

backtracking Integrating DPs with a SAT solver then requires that the imperative data structures be backtracked to stay in sync with the SAT solver, which backtracks while searching for a satisfying propositional assignment.

How well do available programming languages and environments meet these needs?

industrial languages Industrial programming languages like Java, C, and C++ are widely used for implementing DPs. These languages are mature, efficient, and widely used; and they are well suited to implementing imperative data structures. Unfortunately, they lack native support for symbolic computation. Backtracking and an interface to a SAT solver can be built in these languages, but in the absence of a standard library for such things, a new DP will need to implement them from scratch. Systems like SVC (around 22K lines of C++) [2], Simplify (around 26K lines of Modula-3) [6], Verifun (around 10.5K lines of Java and 800 lines of C, not counting the SAT solver) [9], and CVC (150K lines of C++, including the SAT solver) [14] are written in these sorts of languages. The fairly hefty line counts are a tribute to the engineering effort required for these systems!

logic programming languages Logic programming languages are good at symbolic manipulation of expressions and backtracking. With some ingenuity, efficient data structures can be implemented in them, particularly if dynamic clauses are supported [1]. Rewriting can also be performed efficiently. It is not clear how the backtracking of a logic programming environment would be integrated with that of an existing SAT solver. This is because the SAT solver and the logic programming engine are both designed to control when backtracking occurs, and it is not clear how to make them cooperate.

ML and variants Languages like ML and its variants are actually probably the best fit for implementing DPs. Expressions can be implemented as a datatype, and the pattern-matching features of the language can then be conveniently used to rewrite expressions. Mutable references can be used for implementing imperative data structures. Backtracking and interfacing to the SAT solver can be done just as for mainstream languages, with the same engineering cost required. The ICS system (around 5K lines of Ocaml for the logical core, and 6K other lines) [8] and the Vampyre DP from Berkeley (around 10K lines of Ocaml; see http://www-cad.eecs.berkeley.edu/~rupak/Vampyre/) are examples.

rewriting languages Pure rewriting languages like Maude [5] support extraordinarily efficient rewriting, but since they lack mutable references, they do not support efficient imperative data structures.

This paper proposes implementing decision procedures in a new programming language, called Rogue, under development by the authors. Rogue is based on an untyped version of the Rewriting Calculus of Cirstea and Kirchner [4]. Rogue adds backtrackable mutable expression attributes and fixes a particular evaluation order of expressions (which the Rewriting Calculus leaves unspecified). An interface to the Chaff SAT solver [10] is also included. The paper describes work in progress to implement a Rogue Validity Checker (RVC) in Rogue. Because Rogue is such a good fit for implementing decision procedures, RVC is on track to be implemented in much fewer lines of code than decision procedures based on mainstream languages. The prediction is that not counting the size of the Rogue interpreter and the SAT solver, the system can easily be implemented in less than 2000 lines of Rogue code.

2 Rogue

This section begins with some example pieces of Rogue code, and an informal explanation of attributes. It then defines the syntax and gives a big-step operational semantics for the Rogue language. It concludes with a description of the backtracking and SAT interfaces.

2.1 Examples

The code in Figure 1 gives a recursive definition of a function `prop.nnfer`. This function performs the familiar operation of converting a propositional formula built using `NOT`, `AND`, and `OR` from propositional variables to negation normal form (NNF).

```
prop.nnfer :=
 (NOT(NOT('f1))       -> prop.nnfer @ 'f1 |
  NOT(AND('f1,'f2)) -> OR(prop.nnfer @ NOT( 'f1),
                           prop.nnfer @ NOT('f2)) |
  NOT(OR('f1,'f2))  -> AND(prop.nnfer @ NOT('f1),
                            prop.nnfer @ NOT('f2)) |
  AND('f1,'f2)      -> AND(prop.nnfer @ 'f1, prop.nnfer @ 'f2) |
  OR('f1,'f2)       -> OR(prop.nnfer @ 'f1, prop.nnfer @ 'f2) |
  'x -> 'x);
```

Fig. 1. Rogue code for conversion to NNF

The definition consists of a collection of rewrite rules, separated by `|`. When `prop.nnfer` is applied to an expression E, the rules are tried in order. Applications are written using @. A rule `L -> R` matches iff there is a substitution σ for the variables of L such that $\sigma(L) \equiv E$. Variables are indicated with a single quotation mark. The last rule accepts any expression and just returns it. This default will be used here to translate propositional variables and negations thereof to themselves. The two rules just before the last one are congruence rules, which will apply to binary applications of `AND` and `OR` to transform their immediate subexpressions.

As a second example, here is a definition of a non-strict if-then-else operator:

```
u.ite := ('cond -> (1 -> 'then) -> (1 -> 'else) ->
            ((1    -> 'then) |
             (null -> 'else)) @ 'cond)
```

The definition makes `u.ite` take in three arguments: a condition (`'cond`), a then part (`1 -> 'then`) and an else part (`1 -> 'else`). We rely on the fact that in Rogue, evaluation does not occur on the right hand side of an arrow expression. This means that we can delay evaluation of then and else parts by putting them on the right hand sides of arrow expressions. The choice of 1 for the left hand sides of those arrow expressions is arbitrary. After taking in the three arguments, the definition says that depending on whether the condition has evaluated to `1` or `null`, an application of `u.ite` will return either `'then` or `'else`. Notice that whichever of these is returned, it will then be evaluated, because it is no longer on the right hand side of an arrow.

Finally, here is a definition of a `let` operator for local definitions:

```
u.let := ((1 -> ('X,'E1,'E2)) -> ('X -> 'E2) @ 'E1)
```

Evaluating `u.let @ (1 -> ('q,33,'q+'q))` gives 66. This definition of `u.let` is based on the standard translation of let expressions into applications of functions. We use the same trick as for `u.ite` of hiding things on the right hand side of expressions to delay their evaluation. This is necessary to prevent the body of the let expression (`'E2`) from being evaluated before the value of `'E1` has been substituted for `'X`. Our definition is not entirely perfect, however. The standard translation of let expressions is applied as a pre-processing step before evaluation. We do not have such fine-grained control of evaluation here that we can specify such a requirement. Hence, `u.let` expressions will be translated as part of evaluation. This means that `u.let` expressions binding the same variable should not be nested. Nevertheless, `u.let` is still quite useful for making local definitions more readable.

2.2 Attributes

Rogue supports reading and writing attributes of expressions. For example,

```
a.b := c
```

sets the `b` attribute of expression `a` to hold expression `c` as its value. Evaluating `a.b` subsequently (with no intervening writes to `a.b`) returns `c`. We view `X.Y := Z` as an application of a single ternary operator for updating the value of attribute `Y` of expression `X` to store value `Z`. Occurrences of `X.Y` in any other position than immediately on the left of `:=` are parsed as applications of a binary attribute lookup operator.

Any expression in normal form can be used as an attribute of any other such expression. So things like

```
(a -> a).(b -> b) := (c -> c)
```

are allowed. Evaluating $E_1.E_2$ when the E_2 attribute of E_1 has not been set to hold anything returns `null`. The definitions given in the previous section are all just writes to attributes; for example, the `let` attribute of expression `u`. Recursion is supported, since evaluation is not performed on the right hand side of an arrow expression. Thus, if we have

```
foo.bar := ('x -> foo.bar @ 2*'x)
```

then an application like `foo.bar @ 3` will partially evaluate to `foo.bar @ 6`. Even though `foo.bar` was not defined at the point when its definition was being evaluated, it is defined now. So `foo.bar @ 6` will partially evaluate to `foo.bar @ 12`, etc. If evaluation occurred on the right hand sides of arrow expressions, then the definition of `foo.bar` would partially evaluate to

```
foo.bar := ('x -> null @ 2*'x)
```

which would evaluate to

```
foo.bar := ('x -> null)
```

which is not what we want.

As a final note, the mapping from attribute lookup expressions $E_1.E_2$ to stored values is implemented with a hash table, so looking up an attribute takes expected constant time.

2.3 Syntax and Operational Semantics

Rogue has the following syntax. The set of expressions is defined to be the least set containing constant expressions (identifiers and a special constant `null`) and variables (written with a preceding single quote, like `'x`), and closed under forming expressions with binary operators "@" (application), "$\rightarrow$" (formation of a rewrite rule; rewrite rules can alternatively just be viewed as functions), "|" (disjunctive sequencing of rewrite rules), "," (conjunctive sequencing), and "." (attribute lookup); and ternary operator ":=" (attribute update). Numerals of unbounded length and applications of the standard arithmetic operators (written in infix notation) are also expressions.

Figure 2 gives the operational semantics. In order to model the reading and writing of attributes, we make use of stores, which map attribute lookup expressions $E_1.E_2$ to stored values for the attributes. We derive sequents of the form $S :: X \Longrightarrow S' :: X'$, with the intended meaning that if X is evaluated in store S, it evaluates to X' and the store becomes S'. The rules (null-apply), (|-apply-1), (|-apply-2), (,-apply), ($\rightarrow$-apply), and (apply-cong) tell how to evaluate applications. The rules (attr-write) and (attr-read) are for evaluating writes and reads of attributes.

The helper functions used in Figure 2 are defined as follows. Given input E_1, E_2, the function *dropnull* returns E_i if E_{3-i} is `null`, and E_1, E_2 otherwise. Given inputs X, Y, Z, the function *trans* returns $\sigma(Y)$ if there exists a substitution σ for the variables of X such that $\sigma(X) \equiv Z$; and `null` if there is no

$$\text{(base)}\ \frac{}{S :: X \Longrightarrow S :: X}\quad X \text{ a variable or constant}$$

$$\text{(,-cong)}\ \frac{S :: X \Longrightarrow S_1 :: X' \qquad S_1 :: Y \Longrightarrow S_2 :: Y'}{S :: X, Y \Longrightarrow S_2 :: \mathit{dropnull}(X', Y')}$$

$$\text{(|-cong)}\ \frac{S :: X \Longrightarrow S_1 :: X' \qquad S_1 :: Y \Longrightarrow S_2 :: Y'}{S :: X|Y \Longrightarrow S_2 :: X'|Y'}$$

$$(\to\text{-cong})\ \frac{S :: X \Longrightarrow S' :: X'}{S :: X \to Y \Longrightarrow S' :: X' \to Y}$$

$$\text{(apply-cong)}\ \frac{S :: F \Longrightarrow S_1 :: F_1 \qquad S_1 :: Z \Longrightarrow S_2 :: Z_1}{S :: F@Z \Longrightarrow S_2 :: F_1@Z_1}\quad \begin{array}{l} F_1 \text{ not a ``}\to\text{'',} \\ \text{``|'', or ``,'' expr.}\end{array}$$

$$\text{(null-apply)}\ \frac{S :: F \Longrightarrow S_1 :: \mathit{null} \qquad S_1 :: Z \Longrightarrow S_2 :: Z'}{S :: F@Z \Longrightarrow S_2 :: \mathit{null}}$$

$$\text{(|-apply-1)}\ \frac{\begin{array}{l} S :: F \Longrightarrow S_1 :: X|Y \qquad S_1 :: Z \Longrightarrow S_2 :: Z_1 \\ S_2 :: X@Z_1 \Longrightarrow S_3 :: Z_2 \end{array}}{S :: F@Z \Longrightarrow S_3 :: Z_2}\quad Z_2 \not\equiv \texttt{null}$$

$$\text{(|-apply-2)}\ \frac{\begin{array}{l} S :: F \Longrightarrow S_1 :: X|Y \qquad S_1 :: Z \Longrightarrow S_2 :: Z_1 \\ S_2 :: X@Z_1 \Longrightarrow S_3 :: \texttt{null} \qquad S_3 :: Y@Z_1 \Longrightarrow S_4 :: Z_2 \end{array}}{S :: F@Z \Longrightarrow S_4 :: Z_2}$$

$$\text{(,-apply)}\ \frac{\begin{array}{l} S :: F \Longrightarrow S_1 :: X, Y \qquad S_1 :: Z \Longrightarrow S_2 :: Z_1 \\ S_2 :: X@Z_1 \Longrightarrow S_3 :: Q_1 \qquad S_3 :: Y@Z_1 \Longrightarrow S_4 :: Q_2 \end{array}}{S :: F@Z \Longrightarrow S_4 :: \mathit{dropnull}(Q_1, Q_2)}$$

$$(\to\text{-apply})\ \frac{\begin{array}{l} S :: F \Longrightarrow S_1 :: X \to Y \qquad S_1 :: Z \Longrightarrow S_2 :: Z_1 \\ S_2 :: \mathit{trans}(X, Y, Z_1) \Longrightarrow S_3 :: R \end{array}}{S :: F@Z \Longrightarrow S_3 :: R}$$

$$\text{(attr-write)}\ \frac{\begin{array}{l} S :: X \Longrightarrow S_1 :: X' \qquad S_1 :: Y \Longrightarrow S_2 :: Y' \\ S_2 :: Z \Longrightarrow S_3 :: Z' \end{array}}{S :: X.Y := Z \Longrightarrow S_3[X'.Y' \mapsto Z'] :: Z'}$$

$$\text{(attr-read)}\ \frac{S :: X \Longrightarrow S_1 :: X' \qquad S_1 :: Y \Longrightarrow S_2 :: Y'}{S :: X.Y \Longrightarrow S_2 :: \mathit{lookup}(S_2, X'.Y')}$$

Fig. 2. Operational semantics of Rogue

such substitution. If S is a store, then $S[Q \mapsto R]$ is the store that is the same as S except that expression Q is mapped to expression R. Given inputs S a store and Q an expression, the function *lookup* is defined to return R if S maps Q to R, and `null` otherwise. We omit the operational semantics for the arithmetic operations. The implementation uses the GNU MP library to perform arbitrary-precision arithmetic.

2.4 Backtracking

We expose an interface in the Rogue language to backtrack the store assigning values to attributes. Rogue programs make calls to this interface explicitly to do backtracking. Multiple backtracked stores can exist independently in what we call *contexts*. The interface supports switching between contexts, which takes constant time in the current implementation. We give just the following informal description of the interface. For each context, a history of the store is maintained, organized into scopes. When the Rogue interpreter starts, it is in an initial context named `default`, whose scope level is 1. Below, v is a number greater than 0, and C is a constant expression :

`CM @ incsl(v)`: Increment the scope level of the current context by v.
`CM @ decsl(v)`: Let l be the scope level of the current context C, and let l' be $l - v$. If l' is greater than 0, set C's scope level to be l', and revert the store to the last form it had when its scope level was most recently l'. If l' is less than 1, do nothing.
`CM @ sl`: Return the scope level of the current context.
`CM @ cur`: Return the name of the current context.
`CM @ switch(C)`: Switch to the context named C. If there is no such context, create one, set its scope level to 1, and switch to it.

2.5 SAT Interface

Rogue has an interface to a single instance of the Chaff SAT solver. This Rogue interface is built using Clark Barrett's SAT API. We have exposed just the bare minimum of this interface needed to implement RVC:

SATSOLV @ AddVar: Return a fresh propositional variable
SATSOLV @ AddClause(L): Add the clause corresponding to L to the clause database. L is a comma-separated list of literals, where a literal is either a variable `v` returned by SATSOLV @ AddVar, or else a Rogue expression of the form `NOT(v)` for such a `v`.
SATSOLV @ AssignHook(E): Whenever the SAT solver sets a propositional variable x to have a boolean value v, the Rogue expression $(E \,@\, x \,@\, v)$ will be evaluated by the Rogue interpreter. This allows Rogue programs to take some action whenever the SAT solver sets a variable.

SATSOLV @ LevelHook(E): Whenever the SAT solver changes its scope level by some integer i, the Rogue expression $(E \,@\, i)$ will be evaluated. This allows Rogue programs to take some action in this case, like calling one of the CM methods described above.

SATSOLV @ Satisfiable: Return SATISFIABLE or UNSATISFIABLE depending on whether or not the SAT solver finds a satisfying assignment for its clause database.

3 RVC

This section briefly describes the work in progress to implement a Nelson-Oppen combination of decision procedures in Rogue. To check a quantifier-free first-order formula F for satisfiability with respect to a combination of background theories, F is first desugared to remove all propositional operators other than negation, conjunction, and disjunction. It is then *atomified* (cf. [3]). Atomification replaces each distinct atomic subformula ϕ of F by a new propositional variable P_ϕ. A mapping is maintained from ϕ to P_ϕ and vice versa using an attribute called `expr`. One direction of this mapping makes it easy to replace identical subformulas by the same propositional variable. The use of the other is described next.

The resulting abstraction of F is put into CNF in a standard way. Its clauses are then given to Chaff via the `SATSOLV @ AddClause` method. `SATSOLV @ Satisfiable` is called to check the satisfiability of the clauses. Rogue callbacks are registered with Chaff using `SATSOLV @ AssignHook` and `SATSOLV @ LevelHook`. Whenever a propositional variable P_ϕ is assigned a value, the registered AssignHook callback looks up ϕ from P_ϕ using the `expr` attribute. It then asserts ϕ to the decision procedures. If the decision procedures discover that the current set of asserted atomic formulas is unsatisfiable, they communicate this back to Chaff by asserting the disjunction of the negations of the corresponding propositional variables. This causes Chaff to backtrack. Rogue code for desugaring, atomifying, converting to CNF, and making use of the Chaff interface is complete. It is less than 100 lines long.

An online version of the Downey-Sethi-Tarjan [7] congruence closure algorithm is complete. It is about 300 lines of Rogue code, including the code in Figure 3 for the union-find data structure. This code implements path compression and union by rank to get the almost constant amortized running time. Union by rank is implemented even while ensuring that the representative of `union`'s first argument (`'y`) becomes the new representative. This is done using a `rep` attribute. Attributes are also used to hold the ranks and to implement the find pointers. The rest of the congruence closure algorithm uses attributes to hold signatures and signature representatives.

A decision procedure for linear real arithmetic is about 75% complete. It currently stands at a bit under 250 lines of Rogue code implementing canonization of arithmetic expressions and Fourier-Motzkin variable elimination. Attributes can be used for things like mapping an arithmetic variable to the lists of its upper and lower bounding inequalities.

```
uf.union :=
('y -> 'x ->
  u.let @ (1 -> ('fx, (uf.find_top @ 'x),
  u.let @ (1 -> ('fy, (uf.find_top @ 'y),
  u.let @ (1 -> ('rfx, u.if @ (u.not @ 'fx.rank) @
                        (1 -> 'fx.rank := 0),
  u.let @ (1 -> ('rfy, u.if @ (u.not @ 'fy.rank) @
                        (1 -> 'fy.rank := 0),
                 u.ite @ ('rfy < 'rfx) @
                   (1 -> ('fy.find := 'fx, 'fx.rep := 'fy.rep)) @
                   (1 -> ('fx.find := 'fy,
                          u.if @ (u.eq @ 'rfx @ 'rfy) @
                           (1 -> 'fy.rank := 'rfy + 1)))))))))))),

uf.find_top  :=
('x ->
  u.let @ (1 -> ('top, (u.ite @ (u.not @ 'x.find) @
                         (1 -> 'x) @
                         (1 -> 'x.find := (uf.find_top @ 'x.find))),
  u.let @ (1 -> ('tmp, u.if @ (u.not @ 'top.rep) @
                        (1 -> 'top.rep := 'top),
                 'top))))),

uf.find := ('x -> (uf.find_top @ 'x).rep);
```

Fig. 3. Implementation of union-find in Rogue

4 Future Work

One of the biggest concerns with the proposed approach is whether or not the Rogue code can be executed efficiently enough, both in terms of time and memory. Memory is an issue, because expressions with attributes implementing imperative data structures should not be garbage-collected. Currently, this constraint is observed simply by not garbage collecting any expressions, but this clearly will waste unacceptable amounts of memory on long runs. Also, the Rogue interpreter is currently implemented using infrastructure from the CVC system. It should be possible to implement a much smaller and perhaps more efficient interpreter by specializing various data structures to just what is needed for Rogue. It should be possible to use term indexing [12] to improve the performance of evaluating applications of sets of rules connected by $|$.

References

1. A. Appel and A. Felty. Dependent Types Ensure Partial Correctness of Theorem Provers. *Journal of Functional Programming*, 2002. to appear.
2. C. Barrett, D. Dill, and J. Levitt. Validity checking for combinations of theories with equality. In M. Srivas and A. Camilleri, editors, *Formal Methods In Computer-Aided Design*, volume 1166 of *LNCS*, pages 187–201. Springer-Verlag, 1996.
3. C. Barrett, D. Dill, and A. Stump. Checking Satisfiability of First-Order Formulas by Incremental Translation to SAT. In *14th International Conference on Computer-Aided Verification*, 2002.
4. H. Cirstea and C. Kirchner. The Rewriting Calculus - Part I. *Logic Journal of the Interest Group in Pure and Applied Logics*, 9:363–399, May 2001. Also available as Technical Report A01-R-203, LORIA, Nancy (France).
5. Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2), 2002.
6. D. Detlefs, G. Nelson, and J. Saxe. Simplify: the ESC Theorem Prover. *Draft Paper*, 1996.
7. P. Downey, R. Sethi, and R. Tarjan. Variations on the Common Subexpression Problem. *Journal of the Association for Computing Machinery*, 27(4):758–71, 1980.
8. J. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: integrated canonizer and solver. In G. Berry, H. Comon, and A. Finkel, editors, *13th International Conference on Computer-Aided Verification*, 2001.
9. C. Flanagan, R. Joshi, X. Ou, and J. Saxe. Theorem Proving using Lazy Proof Explication. In W. Hunt and F. Somenzi, editors, *15th International Conference on Computer-Aided Verification*, 2003.
10. M. Moskewicz, C. Madigan, Y. Zhaod, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *39th Design Automation Conference*, 2001.
11. A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning*. Elsevier and MIT Press, 2001.
12. R. Sekar, I. Ramakrishnan, and A. Voronkov. *Term Indexing*, chapter 26. Volume 2 of Robinson and Voronkov [11], 2001.
13. R. Shostak. Deciding combinations of theories. *Journal of the Association for Computing Machinery*, 31(1):1–12, 1984.

14. A. Stump, C. Barrett, and D. Dill. CVC: a Cooperating Validity Checker. In *14th International Conference on Computer-Aided Verification*, 2002.